AD-A094 485 GENERAL RESEARCH CORP SANTA BARBARA CA F/G 9/2
AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION--ETC(U)
1980 D M ANDREWS, J P BENSON F49620-79-C-0115
UNCLASSIFIED AFOSR-TR-81-0057 NL

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 19 AFOSR-TR-81-0057	2. GOVT ACCESSION NO. AD-A094483
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dorothy M. Andrews & Jeoffrey P. Benson	8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0115
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Research Corporation PO Box 6770 Santa Barbara, CA 93111	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, DC 20332	12. REPORT DATE 1980
	13. NUMBER OF PAGES 8
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE FEB 3 1981 S D B

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

81 02 03 208

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes an automated testing methodology and an experiment performed to determine its effectiveness. The method is to insert in the program to be tested a number of "executable assertions," statements about the program that trigger error signals whenever they are evaluated to be false (violated). A testcase is then developed for the program using actual values of the input variables. When the program is run, a plot is generated of the number of assertions violated versus the input variable values used. The resulting function is called the "error function." Heuristic search algorithms Over

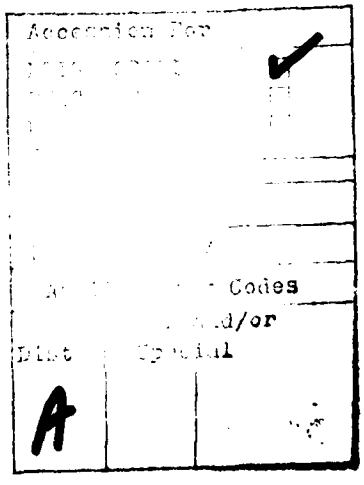
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 402 754

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FILE COPY

can then be used to maximize this function and thereby automatically locate input values which cause the most errors to occur. The experiment included developing asertions for the program to be tested, choosing and inserting representative errors into the program, and implementing search and data collection algorithms for testing. The results indicate that combining executable assertions with heuristic search algorithms is an effective method for automating the testing of computer programs.



AN AUTOMATED PROGRAM TESTING METHODOLOGY AND ITS IMPLEMENTATION

Dorothy M. Andrews* and Jeoffrey P. Benson

General Research Corporation
P.O. Box 6770
Santa Barbara, California 93111

ABSTRACT

This paper describes an automated testing methodology and an experiment performed to determine its effectiveness. The method is to insert in the program to be tested a number of "executable assertions," statements about the program that trigger error signals whenever they are evaluated to be false (violated). A testcase is then developed for the program using actual values of the input variables. When the program is run, a plot is generated of the number of assertions violated versus the input variable values used. The resulting function is called the "error function". Heuristic search algorithms can then be used to maximize this function and thereby automatically locate input values which cause the most errors to occur. The experiment included developing assertions for the program to be tested, choosing and inserting representative errors into the program, and implementing search and data collection algorithms for testing. The results indicate that combining executable assertions with heuristic search algorithms is an effective method for automating the testing of computer programs.

INTRODUCTION

In recent years, an active research area in computer science has been the development of methods for showing that computer programs operate correctly. One result of this research has been executable assertions. If assertions are used to specify the desired behavior of a program, then the program's correctness (relative to the assertions) can be checked automatically. This is done by using the number of assertions violated during a test as a correctness measure for the program. This value indicates whether the program is operating correctly on its input data; the number of assertions violated defines an "error function" over the input space of the program. This removes the need to examine a program's output in detail.

The error function can also be used to automatically generate testcases. It allows standard techniques for maximizing and minimizing functions in multi-dimensional spaces to be applied to the problem of program testing. Automated search techniques such as complex search and heuristic search can be used to find the maximum values of the error function. The input values for which assertions are violated are the input values for which the program fails to work correctly; therefore, it is desirable to find the regions with the maximum violations.

PROBLEMS ASSOCIATED WITH TESTING SOFTWARE

Testing has always been a problem of software development. The method used for testing a program is often a product of the idiosyncracies of the tester. Typically the test criterion is to execute the program for a certain length of time or run a large program through the system. The critical nature of many current software systems makes it imperative to develop a generalized methodology for testing; one that can be applied to many types of programs, thus avoiding the subjective nature of present testing techniques. One way to minimize subjectivity is, of course, to have someone who has not been involved in writing the program do the testing. But this solution in itself brings new problems. The most obvious is that extra time must be allowed to train a new person until he is familiar enough with the program to intelligently make up testcases and procedures for testing. Although computer hardware testing and quality assurance is often a separate department in an organization, this is not commonly the case for software testing.

Testcases

There are two ways to make testing more reliable: increase the number of testcases or make the testcases more specific to the problem. Developing testcases requires much human ingenuity to discover the weak spots in a program and develop input data to test for them. This contributes to the skyrocketing cost of testing since each set of testcases must be tailormade for each software system.

It is important to choose testcases which uncover errors early in the development cycle, not only because the cost of fixing errors increases dramatically with time,[1] but also because of catastrophes that can result from latent errors.

There have been many papers[2-3] on the subject of choosing testcases since it represents one of the most intriguing problems of testing. How does the tester know when enough

input data has been chosen for meaningful testing? In fault-tolerant applications, the test data must include not only the expected values of input data, but also the unexpected values to test for intermittent hardware faults. Since software can have so many states, the number of testcases required increases exponentially.

Test Results

Software testing is unlike hardware testing in that there is no "touchstone" that can be used as a basis for determining if the results from the testing are correct. Test results must be checked manually. In some applications, e.g., ballistic-missile-defense software, checking test results from one run can take several weeks.[4,5] Unfortunately, with software, it is not a matter of determining if a switch is on or off; there is a lot of output to read and analyze.

Last but not least is the psychological aspect of testing that works against a productive testing phase. Once the software is completed, the programmer is anxious to get onto some other project. The challenging and interesting part is designing and implementing the code, not testing it. No one really wants to find errors in his own code, and, furthermore, checking the output is so tedious that it makes the testing process seem routine and boring.

EXECUTABLE ASSERTIONS AND ADAPTIVE TESTING

The major theme that connects most of the problems associated with testing is that of time; it takes time to construct good testcases, time to run them, and time to look at the results. Therefore, one of the ways to address the problem of testing is to automate as much of the testing sequence as possible and to eliminate as much subjectiveness and human intervention as is practical. Fortunately, the basic mechanism to do this, called an Adaptive Tester,[4,5] has been developed over the past several years in response to the need of the Ballistic Missile Defense Advanced Technology Center to develop tests for complex software. The Adaptive Tester is a software system with the following functional components:

- Machine aids for specification of the testing environment
- Automatic preparation of initial test cases
- Automatic performance evaluation
- Adaptive or learning algorithms for selecting test cases

The research effort described in this paper has utilized the components of the Adaptive Tester which generate testcases by automatic perturbation of the input parameters, evaluate past performances of the constructed testcases, and, using this information in a feedback system, generate subsequent testcases. To adapt this powerful capability to this particular application, executable assertions were used as a means of providing data to the performance evaluator. Executable assertions allow the method to be prescribed in general terms and used for any application, since the only thing that varies from one application to another are the assertions themselves.

Executable Assertions

Executable assertions have been found very effective as a simple debugging technique and have been utilized extensively in the development of the Software Quality Laboratory[6] (a large verification system). The primary motivation for adding them was to make debugging easier and quicker since the exact statement number of an assertion that is evaluated to "false" during program execution is stated in a message in the output. For example, if the assertion INITIAL (J .GE. 0 .AND. J .LE. MAXJ), is evaluated as "false", then it is clear that J is negative or it has exceeded the maximum value for J (MAXJ). Without assertions to direct attention to the parts of the program that are not operating as expected, it is often impossible to find the source of the errors that are causing the problems.

Not only are assertions useful for debugging when new code is being added, but the presence of assertions with a special statement to invoke an error recovery routine usually prevents premature termination and allows the program to continue to perform its function.[7-10] Assertions also have proved their worth from the aspect of maintenance and documentation of the system. The Software Quality Laboratory is so large that no one person can be familiar with it all. Assertions which specify the acceptable range of variables help immensely when new code is being written to interface with existing code.

The Adaptive Tester

The Adaptive Tester has been developed in response to the need of the Ballistic Missile Defense Advanced Technology Center to develop tests for software that simulates actual battle conditions. Devising tests in the conventional fashion takes an inordinate amount of time because of the number of parameters that can be varied. Additionally, about a month is required to examine the results of a single run. To get around this situation, ways to perturb the input variables and evaluate the results automatically have been developed. Various search algorithms from artificial intelligence have been implemented to construct new test cases from the results of previous tests.

The search algorithm selected for this experiment employs complex search.[11-14] The technique was developed by Box for solving for the maximum or minimum of a nonlinear function. It involves choosing a set of independent values for the function at random and determining the value of the function for each of these values. The function values define a set of points on a surface called a "complex". Figure 1 gives an example of a complex in three dimensions. The function may have many independent variables, but, for the search routine to function correctly, there must be one more point in the complex than the number of independent variables being perturbed. Assume that the goal is to maximize the value of the function. At each step of the algorithm the point in the complex with the minimum function value is replaced by a new point. The algorithm first attempts to locate the new point on a line connecting the rejected point and the centroid of the other points in the complex.

The distance between the rejected point and the centroid is calculated and the exact location of the new point is then determined. The algorithm has six choices in locating the new point. These are shown in Figure 2 for a complex that is a triangle. Reflection locates the new point at the same distance from the centroid as the rejected point by reflecting the triangle through the centroid. Expansion reflects the triangle through the centroid to locate the new point, but increases the distance between the new point and the centroid. Contraction reflects the triangle through the centroid and reduces the distance that the new point lies from the centroid. Centroid substitution uses the centroid as the new point of the triangle. If none of these operations results in a new point with a function value greater than the rejected point, then two other operations on the triangle can be performed. The triangle can be shrunk by reducing the lengths of one of its sides or rotating it about its centroid.

The coefficients of expansion, contraction, shrinkage and rotation are defined by the user of the algorithm. These operations are performed until a larger function value is found, a predefined limit on the number of operations is reached, or the function attains a value predefined as the maximum value that the algorithm is to locate.

METHODOLOGY FOR ADAPTIVE TESTING WITH ASSERTIONS

The testing methodology used in the experiment is very similar to that used in the Adaptive Testing project. The test configuration consists of several distinct software subsystems: a test driver, the Adaptive Tester, and an assertion evaluator. The program being tested is called the test object. The architecture of the software is shown in Figure 3.

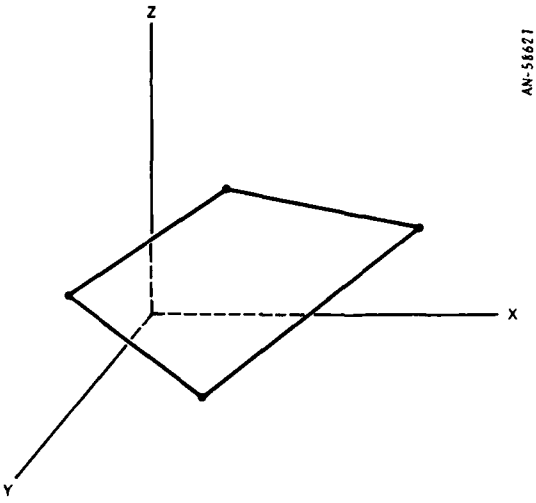


Figure 1. A Complex in Three Dimensions

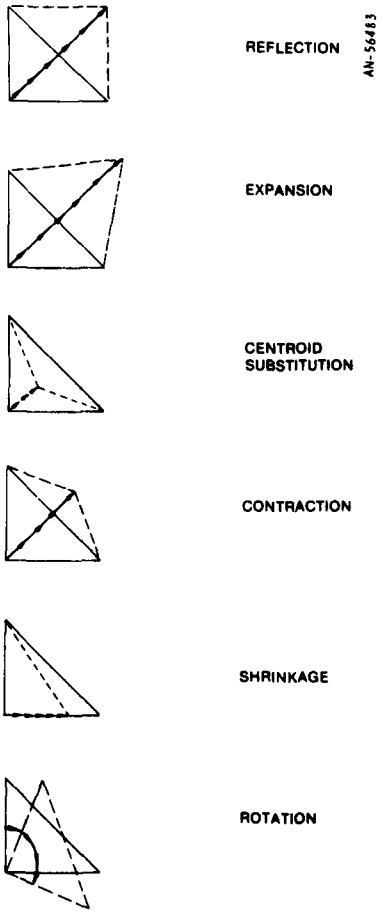


Figure 2. Complex Transformations

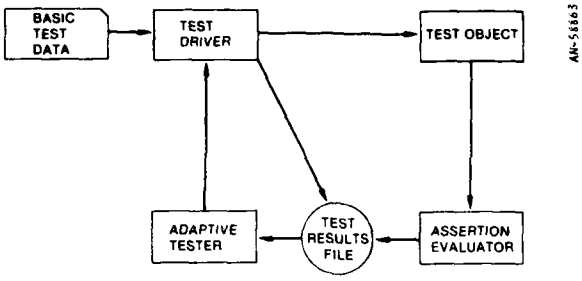


Figure 3. Software Architecture

To initiate the testing process, the tester must specify the following data:

- The input parameters to be altered by the search algorithm
- A set of initial values for each input parameter
- The range of values that each input parameter may assume, in other words, the constaints for each variable
- The maximum number of assertion violations expected

The function of the test driver is to read these values, initiate the testing process, and interface with the Adaptive Tester.

The function of the assertion evaluator is to maintain a test results file containing the following information:

- The value of each input parameter for every test
- The number of assertion violations and the number of different assertions violated
- The statement and module number where each violation occurred
- How many times each assertion was violated

The information in the test results file is used as input to the search algorithm so it can find values of the input variables which cause the maximum assertion violations. The search algorithm constructs a new test case using the operations described in the previous section and returns control to the test driver which executes the next test. At the conclusion of the tests, the test results are output in a final report.

The test object is the program to be tested; it must contain executable assertions. Since they are useful throughout the entire software cycle,[7,10] they should be included in the code as it is being written. This allows the correctness of the assertions to be validated as the code is tested. Since an existing program was used in the experiment, it was necessary to write assertions and then execute the program to be sure the assertions were correct.

Some assertions, such as range checks, are simple to write and do not require in-depth familiarity with the algorithm. For example, in a DO-loop with a variable as the upper bound, it is easy to write an assertion which specifies that the value of that variable is greater than zero. More difficult to write are assertions which check results of computations or that express a relationship between the variables. Since it is necessary to have a firm understanding of the program to write these assertions, it is generally best for the person who implements the code to write these assertions. The success of this testing technique depends on having a sufficient number of assertions expressing tight bounds on variables, thereby enabling them to detect errors.

Once the assertions have been written, the only other part of the testing process requiring human intervention is setting up the first testcase. The remaining part of the testing is automatic: the performance is evaluated and new testcases are generated until a given performance value is attained.

EXPERIMENTS

Two experiments were performed to determine the usefulness of executable assertions in testing. The purpose of the first experiment was to determine if executable assertions could locate errors; and, if so, what the resulting error space looked like. The Adaptive Tester was not used in the first experiment; instead the values of the input parameters were methodically stepped-up in regular intervals across a "grid." The first experiment has been described elsewhere,[15] but the results indicated that executable assertions were effective in detecting errors and that the error function did not include singularities.

The prominent research issues for the second experiment were as follows:

- Behavior of the error function - Does it confirm the results obtained in the first experiment?
- Applicable search techniques - Pending determination of the behavior of the error function, what search technique is the most effective in finding errors?

- Application to large input spaces - What happens when there is a large number of input parameters?

The second experiment was more comprehensive since it actually combined the Adaptive Tester with the use of executable assertions. Since one purpose of this experiment was to provide corroborative evidence of the first experiment, a new test object was selected. The function of the new program was to input an orbit described by a set of eight parameters (orbital elements) and produce a state vector representation of a point on the orbit. Since this program had been in use for twelve years and had been the test object in another experiment[17] it was assumed to be error free. Yet, once the assertions were added to the program, they uncovered latent errors that were completely unsuspected! In most cases, these were errors that occurred only at boundary conditions.

In this second experiment, three modes of operation were implemented in the test driver:

Grid -
The values of the input parameters were varied in a uniform pattern, a grid, over the input space. The results from these grid tests were used as a baseline by which to evaluate the search technique.

Search -
Given one value for each of the tested inputs, the search algorithm constructed all subsequent test cases.

Grid and Search -
Instead of constructing the initial points on the complex from random testcases, a set of values for each of the inputs was derived for input to the search algorithm. These values were derived by sorting on the number of assertion violations obtained from the grid tests; the input values associated with the highest number of violations were passed to the search routine

In each mode of operation, three variables were varied: MODE, VALUE, and the eccentricity of the orbit. To examine the effect of varying a large number of input parameters, additional tests were run in which all ten of the input parameters were perturbed. The Adaptive Tester was able to construct test cases and even found another assertion violation.

Error Seeding

Errors were generated for the test program using a procedure developed by Brooks.[18] The method uses error types and frequencies from a previous study[19] to randomly select a set of errors to be "seeded" in the program. Some types of errors were not chosen for the study, such as documentation, data definition, etc., because the experiment was specifically concerned with detecting run-time errors. The types of errors used were computational errors, logic errors, data handling errors, and interface errors. In generating errors for the experiment, statement types and other descriptive information about the test program were generated automatically using the Software Quality Laboratory. Each statement in the program was classified by type, and a table matching error catagories to statement types was constructed. This resulted in a list of available error sites. Potential error sites were then randomly selected. Once the assertions were written and checked out, errors were introduced one at a time to determine how effective adaptive testing using assertions was in detecting errors.

For each error, a grid test was run. Then tests using the automated search technique were run to see if the results were the same. The search technique was used in two ways: first by perturbing three variables, MODE, VALUE, and one other variable; and then by perturbing all the variables of the orbit at one time. The search routine was allowed to run until it found a preset number of assertion violations (representing the performance value); then this number was automatically stepped up by one and the search algorithm tried to find another combination of input values which would cause a greater number of violations to occur. In this way, the performance value was maximized. The testing process was arbitrarily set to terminate when one hundred tests were run, but each test actually consisted of several subtests because the values of MODE and VALUE were varied within each test. The report that is produced at the conclusion of the runs is shown in Figure 4. In this test MODE, VALUE, and one other variable, ORBIT(6), the eccentricity, were varied.

Test Results

The results of the experiment demonstrated the effectiveness of the assertions in detecting errors. Of the original 24 errors, nine (38 percent) were detected by original assertions, and eight (33 percent) were detected by assertions that were added after the grid testing. There were seven errors that could not be detected by assertions. The reasons why these errors were not detected by assertions were:

- The seeded error was in a section of code that was only traversed after an error occurred.
- Assertions could not be written for some types of errors that were seeded. These included a misspelled variable name, a REAL variable declared as INTEGER, and the wrong number of arguments in a subroutine call.
- The FORTRAN compiler at this installation initializes all variables to zero, therefore a deleted DATA statement caused no problems.

********** FINAL REPORT **********

#RUN	INPUT1	#FALSE ASSERTION	#DIFFERENT ASSERTION	MODE	VALUE
7	0.7526	2	2	4	2477545.659
9	.6048	2	2	5	9849931.060
12	.2700	2	2	4	13958923.49
13	.900	1	1	5	24389119.03
24	.2899	2	2	4	8871067.739
25	.3879	2	2	5	1760571.330
30	.2910	2	2	5	20758872.74
34	.7346	1	1	4	22330022.80
35	.1852	2	2	5	27015515.91
37	.3555	2	2	4	19513234.41
44	.6973	2	2	4	4044234.171
45	.6235	2	2	5	0.
47	.5851	2	2	4	4533190.345
49	.9000	2	2	5	0.
51	.7234	2	2	4	4533190.345
53	.9000	2	2	5	5737662.000
55	.7234	2	2	4	7402021.345
63	.7053	2	2	5	0.
65	.6261	2	2	4	4533190.345
73	.7474	2	2	5	0.
75	.6471	2	2	4	4533190.345
83	.8071	2	2	5	0.
85	.6769	2	2	4	4533190.345
86	.1774	1	1	4	23124989.06
87	.9000	2	2	5	1415222.244
89	.7228	2	2	4	5588024.749
90	.9000	2	2	5	1939816.571
91	.1557	2	2	4	6107643.223
92	.5503	2	2	4	9951144.293
93	.3530	2	2	4	8029393.758
94	.4955	2	2	4	11674092.79
95	.8433	1	1	5	18860857.79
96	.5648	2	2	4	11115483.73
97	.7040	1	1	4	14988170.76
98	.6108	1	1	4	17228371.99
99	.2554	2	2	5	3876077.474
100	.5485	2	2	4	11161715.66
101	.4019	2	2	5	7518896.567
102	.1000	2	2	5	7331062.939

INPUT1 = ORBIT(6)

MODULE	STMT#	TYPE	FAILURES*
ORBP	109	ASSERT	34
OUTCHK	142	ASSERT	38

* HOW MANY RUNS EACH ASSERTION FAILED IN 102 RUNS

Figure 4. Summary of Search Testing for Error 3

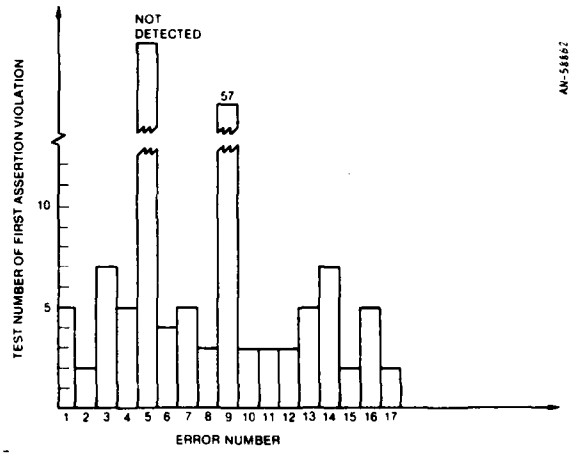


Figure 5. Efficiency of Search Method

Some of the errors that assertions could not detect would have been detected by static-analysis tools which test the consistent use of variables.

For all but four of the errors, the search methods detected the same errors as the grid tests; but they were able to do so much more efficiently and used much less computer time. Figure 5 shows the efficiency of the search technique when all variables are varied; it plots the number of the test in which the first assertion violation was detected versus the error number. Fifteen of the seventeen errors were detected within the first seven tests devised by the search technique. In contrast, the grid technique was run for 317 tests and discovered all but one of the errors; but 683 tests had to be run to detect error number 11.

CONCLUSION

One of the themes emphasized at recent conferences is that new methods for system development and testing are necessary. The need to make software less labor intensive must result in new automated programming tools.

The results from this experiment indicate that this automated testing technique has the potential for finding errors (logic, computational, etc.) that are difficult to find in other ways. In addition, the search algorithm eliminates the subjectiveness in constructing testcases and increases the variety of input values. By automating the testing process, the cost of testing can be reduced dramatically.

Combining executable assertions with adaptive search has resulted in a tool which allows more automation of the software development process and a more accurate testing environment by which to provide software reliability.

ACKNOWLEDGEMENT

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under Contract F49620-79-C-0115. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

REFERENCES

1 Tomlinson G. Rauscher, "A Unified Approach to Microcomputer Software Development", Computer Magazine, June 1978.

2 John B. Goodenough, Susan L. Gerhart, "Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

3 W. E. Howden, "Theoretical and Empirical Studies in Program Testing," IEEE Transactions on Software Engineering, Vol SE-4, July 1978.

4 D. W. Cooper, "Adaptive Testing," Second International Conference on Software Engineering, 13-15 October 1976, San Francisco, CA.

5 D. W. Cooper, Adaptive Learning Requirements and Critical Issues, General Research Corporation CR-4-708, January 1977.

6 J. P. Benson and R. A. Melton, "A Laboratory for the Development and Evaluation of BMD Software Quality Enhancement Techniques," Proceedings of the Second International Conference on Software Engineering, IEEE Computer Society, 1976, pp. 106-109.

7 Dorothy Andrews, "Using Executable Assertions for Testing and Fault Tolerance," 1979 International Conference on Fault Tolerant Computing, Madison, Wisconsin, June 20-22, 1979.

8 Sabina Saib, "Distributed Architectures for Reliability," Proceedings of the AIAA Computers in Aerospace Conference II, Los Angeles, October 1979.

9 Sabina Saib, "Verification and Validation of Avionics Simulation," Proceedings of the AGARD Avionics Panel on Modeling and Simulation of Avionics and Command, Control and Communications Systems, Paris, France, October 1979.

10 Dorothy Andrews, "Using Executable Assertions for Testing," Proceedings of the 13th Annual Asilomar Conference on Circuits, Systems and Computers, November 1979.

11 M. J. Box, "A New Method of Constrained Optimization and a Comparison with Other Methods," Computer Journal, Vol. 8 (1965).

12 J. A. Richardson and J. L. Juester, "Algorithm 454--The Complex Method for Constrained Optimization", Comm. ACM, Vol. 6, No. 8, August 1973.

13 K. D. Shere, "Remark on Algorithm 454," Comm. ACM, Vol. 7, No. 8, August 1974.

14 K. D. Shere, The Box Optimization Method, Naval Ordnance Laboratory NOLTR-74-167, October 25, 1974.

15 J. Benson, A Preliminary Experiment in Automated Software Testing, General Research Corporation TM-2308, February 1980.

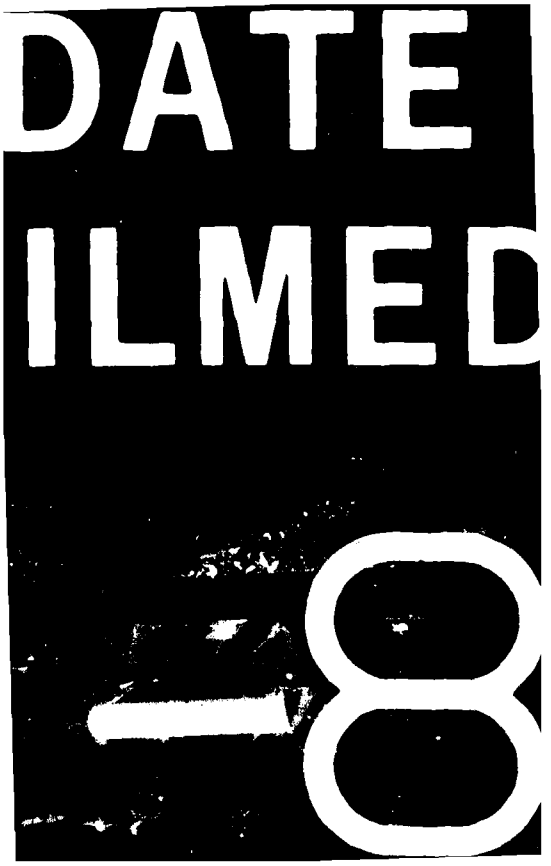
16 T. Plambeck, The Compleat Traidsman, General Research Corporation IM-711/2, revised edition, September 1969.

17 R. N. Meeson, C. Gannon, "An Empirical Evaluation of Static Analysis and Path Testing," Proceedings of AIAA Computers in Aerospace Conference II, Los Angeles, October 1979.

18 N. B. Brooks, An Experimental Evaluation of Software Testing General Research Corporation CR-1-854, May 1979.

19 T. A. Thayer, et al., Software Reliability Study, TRW Defense and Space Systems Group RADC-TR-76-238, Redondo Beach, California, August 1976.

*Dorothy M. Andrews is now at Xerox Corporation, Palo Alto, California.


DATE
ILMED
8